CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Distributed Ada Target Applications**



CIN: **C02 085KU 0001 00**

24 MARCH 1989

COMPASS

Compass, Inc. • 550 Edgewater Drive • Wakefield, MA 01880 • 617-245-9540

# Distributed Ada Target Applications Final Report

by

C. Mugur Stefanescu



CADD-8810-1001
October 10, 1988

revised
February 6, 1989

This work was prepared under contract number DAAB07-85-C-K572 for the Department of the Army, Fort Monmouth, New Jersey.

# Contents

## 0.1 Executive Summary

This is the final report on the investigation into real-time/runtime issues encountered while developing the DAPSE[1] prototypes. This investigation encompasses two reports:

1) "Distributed Ada Target Applications" by C. Mugur Stefanescu.
This report describes the research conducted in DAPSE addressing the methodology and software tool support for development of distributed applications in Ada. It contains an overview of the DAPSE project, a presentation of the general framework for distributed programming and of the DAPSE distribution strategy, and a description of the Distribution Tool Kit (DTK) including a discussion of lessons learned and unresolved issues; AND

2) "Ada Runtime and DAPSE: Experiences, Issues, and Recommendations" by Rowan H. Maclaren, May 17, 1988.
In this report, Ada runtime issues, encountered when implementing the DAPSE prototypes, are identified and discussed. The report concludes with a recommended procedure for evaluating the runtime features of an Ada compiler.

[1]The DAPSE (Distributed Ada Programming Support Environment) research was performed under contract DAAB07-85-C-K572 for the U.S. Army, Ft. Monmouth, N. J.

# Chapter 1

# Distributed Ada Target Applications

## 1.1 Introduction

Ada has been designed to support the development of sequential, parallel, and distributed software applications. As of today, however, none of the existing compilers, runtime environments, or Ada programming support environments provide adequate support for developing distributed Ada programs. The state-of-the-practice in developing distributed Ada applications is to manually distribute distinct Ada programs and implement the required communication between them (e.g., substituting the regular Ada communication mechanisms by system calls to an underlying communication layer).

The DAPSE research project aims at:

- Producing a technology base for developing future Ada programming support environments
- Demonstrating the suitability of this technology base by using it to build a prototype environment for the development of distributed Ada software [10], [11].

The DAPSE project addresses issues related to supporting the cost-effective development and maintenance of quality Ada software [4].

This paper describes those parts of the DAPSE project that specifically address the methodology and software tool support for building distributed applications in Ada. The paper contains a brief overview of the entire DAPSE project and the general framework for distributed programming, a presentation of the specific distribution strategy chosen for the DAPSE project and a description of the initial DAPSE Distribution Tool Kit (DTK) supporting the distribution strategy, and a discussion of research directions and unresolved issues.

## 1.2 DAPSE Project Overview

The primary goals of the DAPSE project are the development of an architectural framework for future APSEs and the development of a coherent environment construction method allowing for customized generation of environments from formal specifications (generator technology). Within this framework, user interface techniques and approaches to distributed programming [15] are addressed. Research results are derived and evaluated through a series of experimental prototypes.

### 1.2.1 DAPSE Construction Method: Generators

The DAPSE technology base allows for the customized generation of automated tools from specifications within the context of an environment framework. This generator approach provides for tailorability and flexibility at the tool level. An instantiation of this approach for the process of compilation is presented in Figure 1.

Tools (e.g., compilers) automate software processes (e.g., translating source code into object code). Before generators (e.g., compiler-compilers) can generate tools from specifications (e.g., grammars), a thorough understanding of the principles underlying the processes and the structure of the corresponding tools is required. Such thorough understanding is usually the result of building and evaluating a series of prototype tools. The actual generator approach is reflected in the horizontal dimension of Figure 1, whereas the preceding phase of understanding and learning is reflected in the building of a number of prototypes in the vertical dimension of Figure 1.

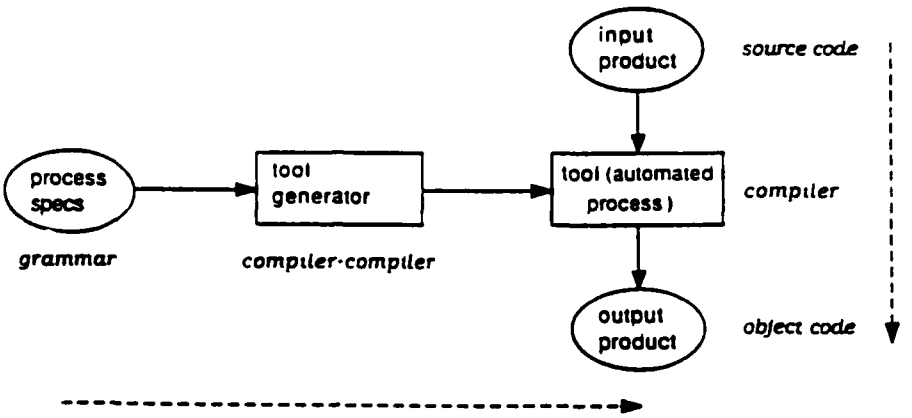



**Fig.1 Generator technology paradigm for compilers**

Generator technology increases the potential for reuse by allowing the reuse of generation of processes in addition to individual tools (the results of such generation processes). This approach decreases the "reuse with modification" problem to modifying just the tool specification instead of the tool itself.

### 1.2.2 DAPSE Objectives with Respect to Distribution

Creating a technology base for developing distributed Ada applications is one of the main objectives of the DAPSE project. The idea is to support the development of distributed Ada applications throughout the lifecycle, especially during specification, design, coding, distribution, and testing. Automated support will be provided through a set of tools called the Distribution Tool Kit. Prototyping these tools should provide insights into applying the generator-based construction methodology to Distribution Tool Kits.

## 1.3 A Distributed Programming Framework

Like any programming language, Ada cannot solve all the problems related to distributed programming. A language generally allows the representation of the result of some problem solving process. The problem solving process itself is concerned with mapping some application into a software solution running on a distributed computing system. This mapping will exploit the inherent parallelism of the application and the resource characteristics of a given computing system to fulfill non-functional requirements (e.g., performance, fault-tolerance). The Distribution Tool Kit conforms with a software process model for developing distributed software, which consists of four stages (Figure 2):

1. **Application development stage** specifies and designs an application solution. During these activities, the inherent application-oriented parallelism is exposed (implicitly or explicitly).

2. **Implementation stage** represents the exposed potential parallelism using the features provided by an implementation language (e.g. Ada). The actual language features, which are supposed to provide a certain level of abstraction, are distinct from their particular implementation by compilers and runtime systems.

3. **Configuration stage** decides on the actual physical distribution and tries to find a good mapping of the parallelism contained in the implemented software product and the physical parallelism provided by a distributed hardware architecture.

4. **Execution stage** deals with all issues related to the actual testing and usage of distributed software in its target environment.

Solutions at these four stages are *not* independent of each other. For example, if Ada is to be used during the implementation stage, the application development stage must be compatibly supported. The distribution model used for designing a solution must allow for the utilization of the features provided by the implementation language.

Insufficient solutions at one stage can be compensated by richer solutions at other stages. For example, lack of appropriate explicit distribution

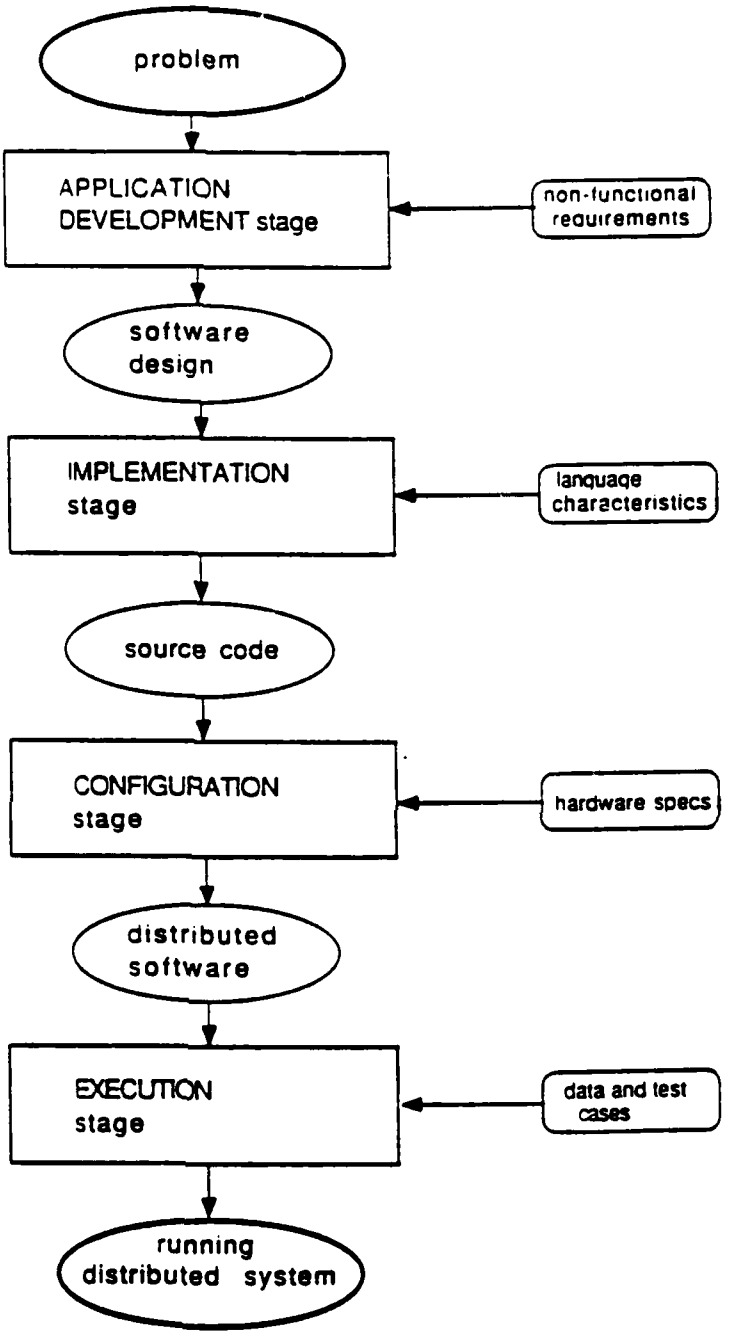



**Fig.2 Four-stage model for building distributed applications**

units in the implementation language can be compensated for by guidelines for distribution (compatible with the unit structure available at the implementation stage) at the application development stage. A similar compensation is made when software is designed in a modular way without having the corresponding features in the implementation language. The solution is to incorporate the idea of modularization (e.g., abstract data types) into the design model and implement them implicitly with whatever features the implementation language offers. For example, the concept of abstract data types can be used whether the implementation language is Ada or Fortran. The advantage of Ada is that there is a corresponding language feature (packages) to reflect abstract data types explicitly. In the same way, although it would be desirable to see some distribution model explicitly incorporated into a language that claims to support distributed programming, Ada can be compensated for by using good distribution models at the application development stage.

### 1.3.1 Characterization of the Approaches

An analysis of past and present distributed systems projects, such as ARGUS, CIRCUS, CLOUDS, EDEN, INCAS, LOCUS and ISIS [14], shows a wide spectrum of potential alternatives for distributed programming. Important related issues are summarized below.

- **What is the appropriate distribution model?**

  **(a) Granularity/type of distribution units** (e.g., entire programs, subsystems = clusters of passive and/or active modules, statements):

  A discussion of various approaches and related consequences in the case of Ada is contained in [Cornhill 84]. Cornhill distinguishes between distributing entire Ada programs, Ada tasks (active units), packages and subprograms (passive units), and all kinds of named program units in the most general case. For example, the distribution of entire programs assumes an early structuring of an application into separate programs. The benefits are that each of those programs can be developed without further considerations of distribution, except for external

communication. The disadvantages are that the entire application cannot be compiled as one program (e.g., no static checking of interface consistency at compile time). On the other hand, any other model of distribution units (except library subprograms and library packages with clearly defined interfaces) creates problems in Ada without guidelines for using it properly (e.g., no sharing of reference variables between distribution units, etc.). There are two solutions to this problem: (i) to restrict the implementation language, or (ii) to provide application development support that guides the structuring of the application towards proper use of concepts that can be represented by explicit language features.

(b) **Type of communication among distribution units**

Synchronous mechanisms are preferred from a software engineering perspective, because they allow proofs of the program correctness. Logically, they extend the traditional local communication mechanisms to remote communication. However, not all kinds of real-time requirements can be fulfilled with this kind of mechanism. The open question is whether both types need to be provided at the language level, or whether synchronous communication at the language level is sufficient, with the actual implementation delegated to an intelligent compiler or runtime system.

- **How should communication between distributed Ada units be provided, transparently or non-transparently?**

Non-transparent communication assumes that the developer must be aware whether an inter-unit communication is local or remote before the loading phase; transparent communication does not. Non-transparent communication might mean that the developers have to choose between local and remote communication features or must implement remote communication themselves. Both of these approaches have the disadvantage that they result in static configurations of a distributed program. Whenever the parts of the distributed program will be re-hosted, some re-programming might be involved. Ada claims to allow for distribution. Currently, the overwhelming majority of

validated Ada compilers do not support transparent remote communication.

Transparent communication assumes that the developer creates a net of communicating software distribution units independent of physical distribution. This can be achieved either by having a compiler that (based on distribution information) creates different code for local or remote communication, a runtime system interpreting communication calls (based on distribution information) during execution, or by a post-processor transforming compiled centralized programs into distributed programs (based on distribution information). All three approaches are possible in Ada.

- **When should the developer deal with the distribution aspects?**

  This is the most crucial part of the distributed programming framework. There are applications that require a certain distribution, because of the configuration of the physical process (e.g., embedded control systems). There are other cases (especially if parallelism and distribution are to be exploited at the statement level or lower) where the user does not have to deal with distribution at all, because an intelligent compiler can derive a good distribution simply based on data-flow analysis. In general, distribution should be dealt with in three steps: (i) identify the potential parallelism of some application, (ii) formulate this parallelism as part of the product implementation, and (iii) map the product onto some distributed hardware architecture, trying to exploit the parallelism reflected in the implementation to the degree possible or desired. Steps (i), (ii) and (iii) are usually dealt with during the application development, implementation, and configuration stages respectively (see Figure 2).

In summary, there are three different approaches to distribution:

1. **Only during the application development stage.**

   Identify units of distribution based on application constraints (e.g., embedded systems) early on, and develop each of these distribution units without regard to distribution.

2. Only during the configuration stage.

   Disregard distribution during the application development and implementation stages and deal with distribution based on an analysis of the implemented product (e.g., in the case of distributing a program at the statement level onto a vector machine).

3. Throughout the application, implementation, and configuration stages.

   Guide the user during the application development stage towards defining distributable units that are compatible with the structural concept of the implementation language. Guide the user during the implementation stage towards using the implementation language features effectively and suggest mappings of distributable software units onto physical hardware nodes during the configuration stage.

### 1.3.2 Necessary Support Areas

In order to effectively implement non-functional requirements, such as performance or fault-tolerance (which are often the drivers for distribution), it is necessary to support the application development, implementation, configuration and execution stages properly. Proper support can address the **constructive** steps for achieving distribution or the **analysis** of constructed products with respect to distribution. For example, to only allow the distribution of non-nested packages, it is necessary to provide either:

- Constructive support in the sense that a design model is followed which does not permit nesting of package-like units
- Analysis of a program, developed without knowing the non-nesting requirement, and identification for distribution of candidate packages which are not nested

There are limitations to supporting constructive aspects of distributed programming due to the poor understanding, as a community, of the process of developing distributed software. Analysis of case studies of actual developments and experiments can help in understanding what needs to be supported during the development of distributed programming. Only when the positive and negative impacts of a chosen distribution approach are understood can development be effectively supported.

### Constructive Support

Constructive support is required for all four stages of the development model (see Figure 2). Non-functional requirements, the drivers for distribution (e.g. fault tolerance, performance, availability, security), are specified during the application development stage. Without traceability of non-functional requirements, what would unit and integration testing mean as far as such requirements were concerned? During the implementation stage, an appropriate language compiler and an editor (structured or syntax-directed) are needed in addition to guidelines for using the existing features of an implementation language in the context of a distribution model. During the configuration stage, guidelines are needed as to what mappings (software onto hardware) are promising to fulfill the non-functional requirements. This includes the specification of hardware characteristics. During the execution stage, support for distributed loading, testing, and monitoring are needed.

### Analytic Support

Analytical support for feedback and control should be provided when construction cannot be automated and for the purposes of learning about the construction process. Examples from the application development stage are the evaluation of designs with respect to possible degree of parallelism, or the checking of the validity of traces of non-functional requirements throughout the design process. Examples from the implementation stage are the analysis of source code to determine whether language features were used properly or whether the implemented program restricts the degree of parallelism more than necessary (compared to the potential parallelism expressed as a result of the application development phase). Examples from the configuration stage are determining a good mapping of a given source code program onto a distributed system, determining a good distributed system architecture for fulfilling specific non-functional requirements with a given source code program, or identifying the implicit potential for parallelism of a given program that does not explicitly address parallelism. Examples from the execution stage are determining the actual performance or fault-tolerance of a distributed system (testing or monitoring software in its target environment), determining the impact of software structure, language features and the implementation of those features

## 1.4 Distribution Strategy of the DAPSE Project

This section describes the approaches taken towards a DAPSE Distribution Tool Kit in the context of the framework presented above. This will allow comparison of the DAPSE approach with those chosen by other research and development groups.

DAPSE is a project that aims in part at creating a technology base for developing distributed Ada applications. As in every research project of this kind, the prototype implementations are not intended as marketable products, but as existence proofs for the kinds of technologies suggested, as well as vehicles for the further evaluation and improvement of those technologies.

The distribution strategy chosen for the DAPSE project is first motivated by the application domain of interest, the particular constraints imposed by an Ada environment on choosing a particular strategy, and the expectations for this project. The specific strategy is presented in the context of the framework presented in section 3, and ideas for each of the support areas are outlined.

The research character of DAPSE creates expectations at various levels: (i) building knowledge as to how distributed programming in Ada should be supported, (ii) lessons learned about the impact (pros and cons) of certain language features, as well as their particular implementation (it is important to distinguish here between Ada language issues and Ada implementation issues!), and (iii) contributing to the better understanding of the issues of distributed programming in general.

### 1.4.1 Application Domain

The application domain at this time is "interactive systems" that do not impose any hard real-time requirements. The performance requirements imposed are that response times should not exceed the limit that can actually be recognized by human beings, on the order of 1/10 second. Again, this application domain will be used to prototype and evaluate alternative approaches. It is possible that at some later time (armed with a better understanding and validated approaches), application domains with harder non-functional requirements could be supported.

### 1.4.2 Rationale

The rationale underlying the DAPSE strategy is:

- To be able to compile a distributed Ada application with existing validated Ada compilers.
- To utilize Ada language features in a way that allows to maximize desirable characteristics (e.g., performance) through distribution.
- To build upon the existing body of knowledge regarding distribution [16].
- To develop a prototype distribution tool kit based on the generator technology.

The last issue suggests the need for further explanation of the generator technology. Figure 3 presents the the generator technology paradigm instantiated for building distributed applications.

The DAPSE investigation proceeds along the vertical line, trying various approaches for the distribution tool kit. As understanding grows (this means that when tools can be structured in a systematic way while the impact of different distribution strategies is understood), it will be possible to generate at least parts of those tools from some specification of the distribution strategy (horizontal line).

### 1.4.3 DAPSE Strategy

Due to the research character of this project, one particular approach is described in this and the following subsections which can be compared with alternate approaches.

The objectives of the DAPSE Distribution Tool Kit are:

- To use all kinds of functional units (subsystems, passive and active modules) as distribution units that are compatible with the structural units provided by Ada
- To use preferably synchronous communication, for the usual software engineering reasons, without ruling out the need for using asynchronous communication

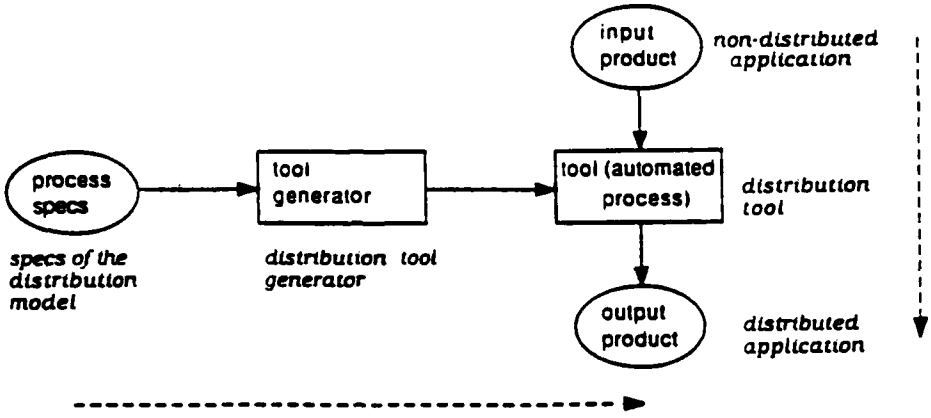



**Fig.3 Generator technology paradigm for distribution tools**

- To provide transparent communication by generating the required code for communication between remote units of distribution using a post-processor which correctly translates compiled centralized Ada (together with a description of its physical mapping onto a distributed computing system) into a distributed Ada program. Transparent communication layers allowing for dynamic reconfiguration of distributed Ada programs could eventually be added.
- To support the development of distributed Ada programs at all four stages of the model

### 1.4.4 Research/Development Plan

The plan for achieving the objectives includes the following tasks:

1. To develop a tool for automatically generating the necessary communication code based upon an Ada program, a description of the distributed architecture, and a description of the intended mapping of the Ada program onto this hardware

2. To explore technology, which could be used to develop a tool, for analyzing the expected performance and/or fault-tolerance of an Ada program based upon its mapping onto a given distributed architecture

3. To explore technology for helping the developer in identifying a "good" mapping of a given Ada program onto a given distributed architecture, or identifying a good distributed architecture for executing a given Ada program ("good" is characterized in this context as satisfying given performance or fault-tolerance requirements)

4. To document the lessons learned, describing the impact of certain characteristics of an Ada program on its distribution potential (these lessons learned should result in constructive guidelines for the developers), in relation to non-functional requirements

5. To develop a comprehensive development methodology for distributed software that covers all phases, from specification to implementation and testing (the important aspects are to allow for the traceability of characteristics that have been identified as impacting the distribution potential through all software representation levels)

6. To provide the appropriate infra-structure (measuring those characteristics) throughout the development process and guiding the developer through appropriate feedback based on past experience

### 1.4.5 Distribution Support Specific to the Life Cycle

Initial ideas as to how the application development, implementation, and configuration stages could be supported are listed below.

- **During the application development stage**, provide support regarding the specification and design towards distributed Ada programs:
  - A specification language to allow the specification of non-functional requirements in addition to functional requirements. It is possible to augment and modify existing specification methods (e.g., TSL [9]).

- A (graphically oriented) design method, compatible with the specification method as well as with Ada, that allows for the traceability of refined non-functional requirements. The idea is to support a step-wise refinement process, including functional and non-functional requirements.

- **During the implementation stage,** apply analysis and constructive guidelines for properly using Ada language features.

- **During the configuration phase,** provide support for specifying the characteristics of a physically distributed hardware architecture, and mapping a given Ada program (structured in terms of distributable units) onto such an architecture relative to specific non-functional requirements. Although this problem cannot be solved in general, practical constraints may limit the domain of possible reductions and thereby make the problem solvable.

- **During the execution stage,** support distributed loading, testing, and monitoring, using as many results as possible from other ongoing research projects

## 1.5 The Prototype DAPSE Distribution Tool Kit

The prototype Distribution Tool Kit implements one specific, limited distribution approach within the scope of the DAPSE distribution strategy.

### 1.5.1 The Initial Approach

The initial approach for the Distribution Tool Kit restricts the DAPSE strategy as follows (See Figure 4):

- To use "library packages and library subprograms" as the candidate units of distribution. A further research issue is the investigation into the possibility and problems of allowing non-library units or even all named Ada units that require no dereferencing to be distributed.

- To use synchronous communication

| Issue | Framework | DAPSE | DAPSE Distribution Tool Kit |
|---|---|---|---|
| Granularity | process/subsystem/ module | subsystems, modules (library units) | library units |
| Communication | -synchronous -asynchronous | -synchronous [-asynchronous] | synchronous (RPC) |
| Communication transparency | -transparent -non-transparent | transparent | transparent (source expansion) |
| Stage | -application development -implementation -configuration -execution | -application development -implementation -configuration -execution | -configuration [-execution] |

**Fig.4 Comparative table of proposed framework and current work in DAPSE**

- To generate communication code by a post-processor as required, based on the physical distribution of an Ada program
- To handle distribution only during the configuration stage

The initial efforts are based on Schuman's approach [16]. This approach is based on Hoare's work on Communicating Sequential Processes (CSP) [7] and assumes that the application is composed of a fixed number of distribution units (mutually disjoint in address space) and communication is done through synchronous typed message passing.

In this approach, application development according to the distribution paradigm consists of an appropriate packaging of the code and specifying the mapping on a real architecture. It calls for providing distribution by transforming certain constructs (procedure calls, entry calls, constant references) into Remote Procedure Calls [12]. The RPCs are implemented by dedicated channels, which provide typed bidirectional message passing.

This first approach allows dealing with the problems of the configuration and execution stages early on. The pros of such an approach are that it will enable rapid development of a prototype that allows the distributed execution of Ada programs (this is required as a basis anyway). The cons

are that the difficult questions are avoided at this time (at the application development and implementation stage) and that Ada programs that are not suited for distribution due to their use of Ada features are not restricted. Later approaches based on the DTK should gradually introduce support for the application development stage and the implementation and execution stages.

### 1.5.2 Summary of Development Efforts for DTK

The development efforts for DTK[1] were done in three phases:

- Building sample applications manually
- Refining the recipe of code generation to meet better the requirements of distributed applications
- Implementing the prototype DTK

The recipe was refined to address the issues of startup and shutdown in a distributed environment as well as particularities in the Sun/UNIX/Verdix environment. The operations of StartUp and ShutDown refer to the operations that are applied to the communication paths between the distribution units. Both operations require appropriate protocols (reliable and general to cover a large range of needs). StartUp was treated minimally and implicitly, by adopting a solution with empirical delays. ShutDown was provided as an explicit operation in three different semantics:

1. Termination of the availability of the service at caller's site. This is the raw version of shutdown. It includes the termination of the virtual connection between the caller and the callee. It handles explicitly turning off application clients.
2. Termination of the availability of the service at callee's site. This is similar to the one above, but aimed at the other side of the communication (to handle explicitly turning off application servers).

[1]The DTK was implemented on a network of Sun workstations, using the Verdix Ada Compiler.

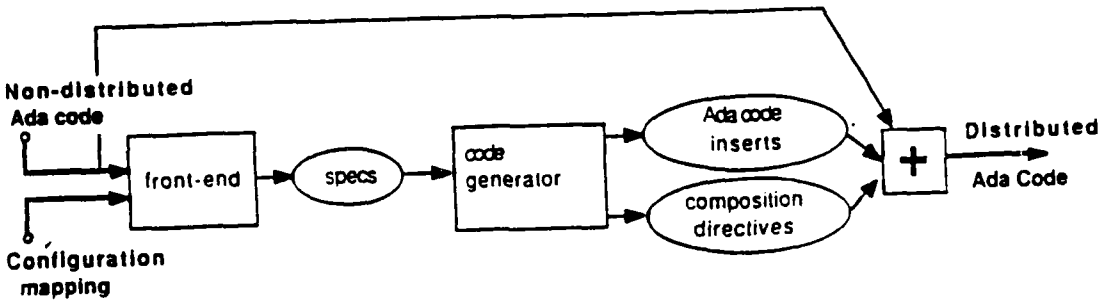



Fig.5 DTK architecture

3. Termination of the availability of the whole virtual connection. This has the semantics of `terminate` in Ada, and provides shutdown of a remote site by a graceful degradation.

DTK takes advantage of existing capabilities in DAPSE such as: Ada structured editor (Appendix A), Ada librarian and Ada parser and unparser. These components allow manipulation of Ada libraries and Ada programs in ASCII and DIANA form.

### 1.5.3 The DTK Architecture

DTK architecture is presented below:

The input consists of Ada code and the mapping onto the desired configuration. The Ada code is in the form a DIANA tree created either with the structured editor or parsed with the Ada parser. The mapping is created with a structured editor similar to the Ada librarian (Appendix B).

The key feature in the DTK architecture is the explicit specification, extracted from the application, of *what must be taken into account with respect to distribution* (similar to the IR in a compiler.) Basically, DTK functions in the following manner:

1. The front-end generates from the Ada code (DIANA) and the configuration mapping, the specs of the application with respect to distribution.

2. The code generator produces (1) the required code inserts for handling communication, (2) the directives of how to combine them with copies of the original code, and (3) scripts for compiling and linking the distributed application.

3. A composing (non-interactive) editor puts the various segments together.

The prototype Distribution Tool Kit consists of a minimal set of small tools capable of performing specialized tasks and exchanging data, allowing for flexibility and extensibility:

- A transformer. This tool expects as input the intermediate representation of the units to be distributed and a specification of the distribution. It will execute the source expansion according to the rules of transformation, called the *recipe*.

- A distribution editor. This is a combined Ada library editor and distribution specifier. It allows specification of the distribution, the mapping of distributable units onto distributed units, and the configuration, the mapping onto the real architecture.

### 1.5.4 Lessons Learned

A number of lessons have been learned from developing the prototype. They can be classified into the following topics:

- Issues related to the initial approach to distribution. The evaluation of the initial approach is done with respect to the appropriateness of the automated source transformation concept and the transformation recipe.

- Experience has proven the validity of the concept of distributing Ada applications based on source transformation. As expected, the process of development is rather flexible for a number of reasons, including delaying the specification of mapping on the actual architecture and small overhead for fine-tuning of the mapping.
- Schuman's recipe [16] is implementable with the current technology. At the time this approach was published (1981), Ada compiler technology was not mature enough for the implementation of the recipe.
- The approach is suitable for a class of applications with static architecture, inherited from CSP[7]. For instance, an example of an unsuited application would be a server with a variable number of clients e.g., a common database accessed by many users running the same application.
- The recipe has to be further refined to completely handle distribution specific issues e.g. startup, shutdown, etc.
- The recipe handles only a minimal set of Ada features. Although this set is probably enough for developing all the applications within its scope, i.e. the set is canonical, additional refinements are needed to take full advantage of Ada power. Exceptions are an obvious candidate.

- Ada and UNIX issues. This topic combines the language issues with the issues related to the interaction between a particular Ada implementation and the operating system.
  - The only language related issue appeared with exceptions in the remote mode, i.e., between distribution units. While it is straightforward to support predefined exceptions, it is very costly to support user defined exceptions, particularly exceptions that cannot be caught explicitly by exception handlers, because they are not visible at that point. Generally, these exceptions are trapped with the **when others** construct and are eventually raised again blindly to the caller. If the call was a remote one, the caller is somewhere on the other side of the virtual connection. Normally, information qualifying the exception is encoded

and returned through the virtual connection for decoding at the caller site. For anonymous exceptions this is not possible, since they are not visible to Ada and special access to the runtime system is needed.

- The interaction between Sun-RPC library and Ada runtime system was rather fragile. The combination of the Sun-RPC routines and obscurely nested Ada constructs, packages with tasks and with nested generic instantiations containing tasks, failed and other less natural programming solutions had to be adopted.
- One of the critical layers dealt with inter-process communication. Several different versions of the communication package were attempted. The version based on sockets proved to be compact and reliable. From the socket types provided by Sun, the datagram connectionless sockets fitted the best. They are the only sockets that allow a flexible protocol of starting up the channels, since there is no need for synchronization of the two ends prior to message sending. The datagram sockets, unlike stream sockets, do not ensure reliability of data transmission, and thus extra checks are needed either at or above the virtual channel level. One possible reliable protocol was described in Schuman's paper.

- Performance issues. Performance and overhead are introduced by distribution:
  - The hand-built examples showed that the approach is well suited for soft real-time applications, where the response time is instantaneous in human terms: hundreds of milliseconds, rather than milli- or microseconds. Further performance improvements might strictly depend on better implementations or special access to the runtime system.
  - The code overhead is given by the Ada code which is automatically generated, and operating system dependent code, C in UNIX, which is generally of fixed size. The Ada code overhead tends to be rather large and increases with the degree of distribution (the number of distributed units) and the number of Ada features supported. This increase measures the generated

distributed program versus the undistributed program rather than the generated distributed program versus a hand-built distributed program. Therefore, the increase is due both to the distribution and to the distribution recipe. When memory constraints are severe, it would be possible to tune the distribution recipe to a particular compiler.

- The number of additional tasks is not really important; the additional tasks put more burden on the task scheduler. The simplest version in Schuman's approach adds only two tasks per process, in the context of UNIX, but the more sophisticated ones add N+2 tasks, where N is the maximum number of concurrent remote calls. This is not expected to be a problem for the class of applications in this domain.

### 1.5.5 Unresolved Issues

The DTK prototype demonstrated the implementability of the DAPSE approach. The prototype serves not only as a proof of concept, but also as a guideline toward construction of a production quality tool kit. The prototype can also be used as an experimental test-bed, such as for gathering statistics on various Ada features or for evaluating a particular distribution configuration. Development guidance is an extension which might be added based on results of statistics gathering experiments. Because the DTK is only a first step in addressing a major research approach, it is important to place the results into context of the unresolved issues and to list possible future directions:

- To explore different communication layer interfaces, particularly seeking more reliable protocols, as discussed in [16]
- To explore enhancements of the distribution model
  1. By extending the scope of Ada features considered until now. In particular, the objects of interest are non-library program units (including tasks) and full semantics for exceptions.
  2. By extending the distribution model beyond the current approach. Currently, a distributed application is confined to having

a completely static architecture. It is particularly interesting to investigate models with more dynamic behavior, for instance, a common database accessed by many users running the same application. In this case, one user is just an instantiated database client, and the number of instantiations is variable, but probably bounded.

- To explore enhancements of the user interface
  1. By extending the support from one (Verdix) Ada program library to an entire library hierarchy
  2. By improving the FENSTER-SEGUE layers
- To complete the minimal Distribution Tool Kit by adding an analyzer. This tool will expect Ada code as input and will check it against the constraints imposed by the distribution recipe.
- To assess the potential limitations of the generator-based technology used in the initial approach

Experimental research issues include:

- Front-end analysis about non-functional requirements such as reliability or performance of the resulting distributed application. This could be implemented using heuristics based on gathered statistics.
- Enhancing the user interface to support better visualization of the distribution
- Debugging support including user interface issues

## 1.6 Conclusions

This paper describes the research and development activities within the DAPSE project aimed at appropriate support for the development of distributed Ada programs. This includes a rationale for the specific DAPSE distribution approach in the context of a general framework for distributed programming. The initial research and development efforts are described and initial lessons learned are summarized. The initial distribution tool kit

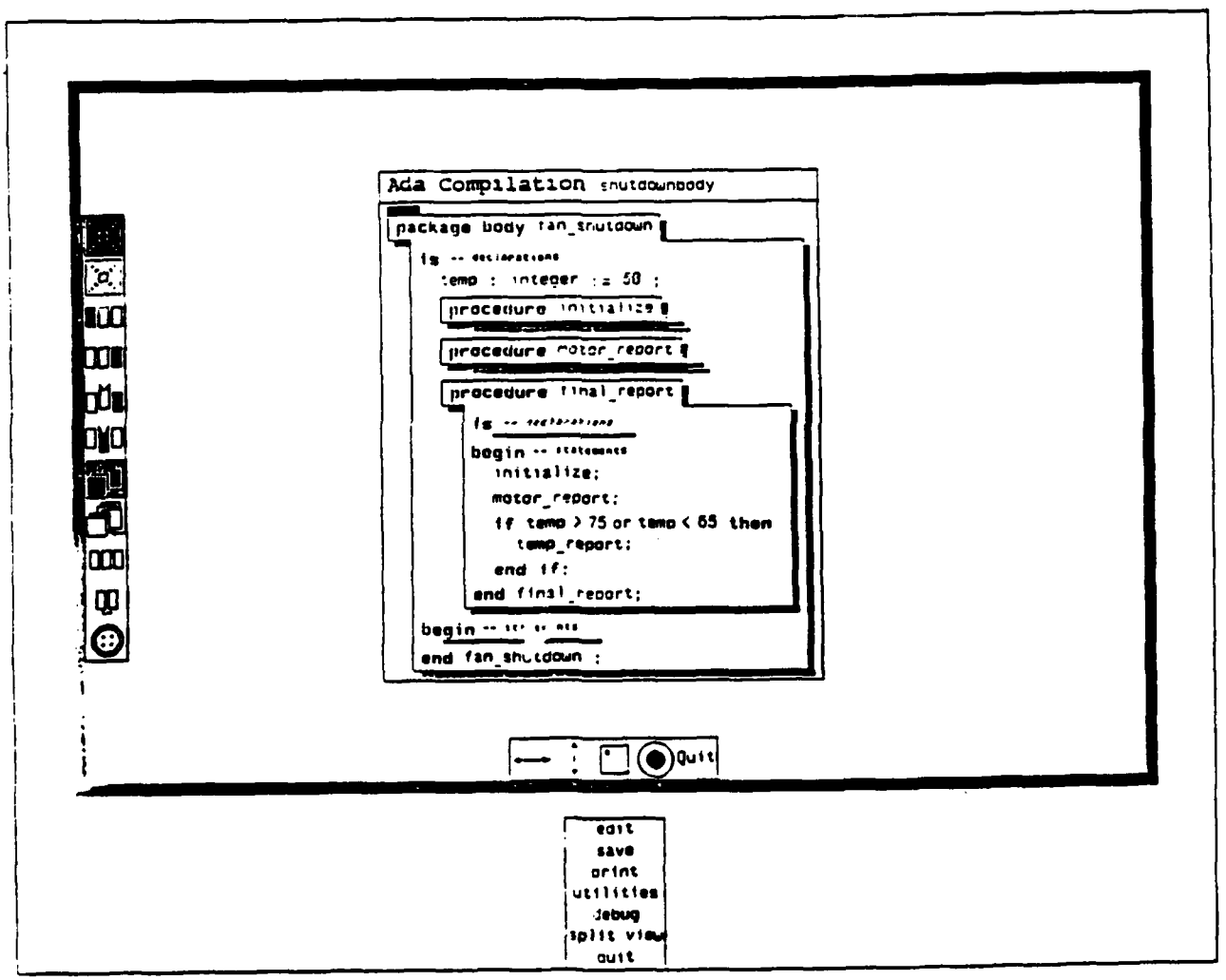
supports a specific approach in the context of the configuration stage of the software process model for developing distributed software.

Continued research based on this work but beyond the scope of the DAPSE project are investigations in three different directions: (1) alternative approaches for communication between units within the context of our current distribution model (library units), (2) alternative distribution models, (3) support for application development, implementation, and execution stages for the most promising distribution models.

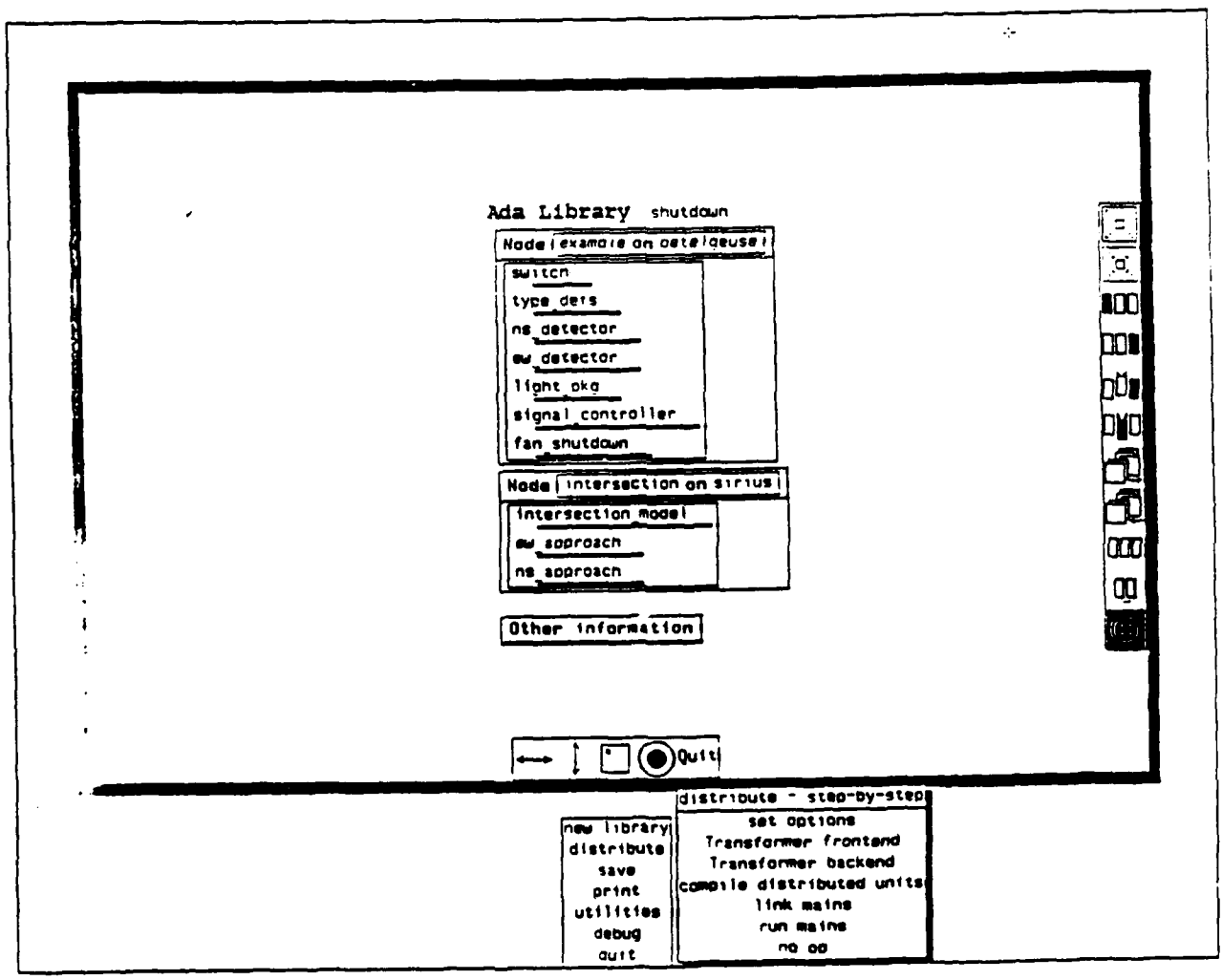
The DAPSE project addresses the need for a customized generation of tools within a context of environment frameworks. The project has the potential of incorporating further customization at the overall process level as in [3], [2]. The key to effectively employing the generator idea for entire software engineering environments is to provide sound specifications of the particular software processes used in a particular organization. Such a specification language could be the basis for deriving tailored specifications for individual environment tool components, customized software engineering databases, and distribution tool components tailored towards a specific distribution strategy.

The body of knowledge required for effectively generating customized programming environments is tremendous. However, the benefits of providing customized programming environments (tailored towards the organization specific processes in a natural way) as far as productivity and quality are concerned promise to be even larger.

## 1.7 Appendix A: The Ada Structured Editor



## 1.8 Appendix B: The Configuration Mapping Structured Editor



# Bibliography


[1] M. R. Barbacci, J. M. Wing, "Specifying Functional and Timing Behavior for Real-Time Applications," Technical Report CMU/SEI-86-TR-4, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, December 1986.

[2] V. R. Basili and H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," in Proc. Fifth National Conference on Ada Technology and WADAS, Arlington, VA, March 1987.

[3] V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," Technical Report TR-1983, Dept. of Computer Science, University of Maryland, College Park, MD, January 1988.

[4] S. Boyd, M. Marcus, K. Sattley, "Extensibility in an Ada Programming Support Environment," in Proc. Sixth National Conference on Ada Technology, March 1988.

[5] E. C. Cooper, "Distributed Systems Technology Survey," Technical Report CMU/SEI-87-TR-5, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, March 1987.

[6] D. Cornhill, "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," in Proc. IEEE 1st International Ada Conference on Ada Applications and Environments, 1984.

[7] C. A. R. Hoare "Communicating Sequential Processes," Prentice-Hall International, 1985.

[8] N. C. Hutchinson, "Emerald: A Language to Support Distributed Programming," in Proc. Second Workshop on Large-Grained Parallelism, Special Report CMU/SEI-87-SR-5, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, November 1987, pp. 45-47.

[9] D. C. Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, and M. A. Haberler, "Task Sequencing Language for Specifying Distributed Ada Systems (TSL-1)," Technical Report CSL-TR-87-334, Computer Systems Laboratory, Stanford University, Stanford, CA, July 1987.

[10] M. Marcus, "DAPSE: A Distributed Ada Programming Support Environment," in Proc. IEEE 2nd International Conference on Ada Applications and Environments, 1986.

[11] M. Marcus, K. Sattley, and C. M. Stefanescu, "Configuration Control in an Ada Programming Support Environment," in Proc. Fifth National Conference on Ada Technology and WADAS, Arlington, VA, March 1987.

[12] Nelson Bruce Jay, "Remote Procedure Call," Carnegie Mellon University, May 1981.

[13] Proc. International Workshop on Real-Time Ada Issues, published as a special Edition of Ada Letters, Vol. VII, no. 6, Fall 1987.

[14] H. D. Rombach, "A Survey of Distributed Systems Projects," presentation, Compass, Inc., July 7, 1987.

[15] S. C. Schaffner, M. Borkan, "SEGUE: Support for Distributed Graphical Interfaces," Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, Vol. II, Hawaii, 1988.

[16] S. A. Schuman, E. M. Clarke, C.N. Nikolau "Programming Distributed Applications in Ada: A First Approach," Presented at the 1981 International Conference on Parallel Programming, CA-8108-1201, Compass, Inc., 1981.

[17] S. A. Schuman and D. H. Pitt, "Object-Oriented Subsystem Specification," in Proc. IFIP Working Conference on Program Specification and Transformation, North-Holland, 1986.

# Chapter 2

# Ada Runtime and DAPSE: Experiences, Issues, and Recommendations

## 2.1 Introduction

In this paper, Ada runtime issues encountered when implementing a DAPSE [1] prototype programming environment are identified and discussed. This is intended to produce a set of criteria useful for evaluating or selecting an Ada compiler in terms of its runtime system. The DAPSE project is a research effort to develop a technology base [1] for Ada programming support environments. As such it makes certain requirements on the runtime system which are not unusually demanding but which take full advantage of the variety of semantic features supplied by the Ada language. Although it is not within the scope of the DAPSE project to assess the runtime systems of all Ada compilers, the specific problems the DAPSE implementors had can be used to identify problem areas and to form the basis of a partial check list for compiler evaluation. This check list could be used to focus the Guidelines for Ada compiler specification and selection prescribed by [5]. The DAPSE experiences are useful because there is not a great deal of information disseminated about Ada runtime systems based on the various

[1]The DAPSE project includes a prototype of a programming environment using the environment development technology.

efforts in developing real-time systems in Ada.

It is important to bear in mind that the Ada compiler technology is immature and therefore runtime problems are to be expected in all Ada compilers. Most compilers have been implemented with emphasis on passing the validation suites. Although the ACVC (Ada Compiler Validation Capability) tests do check runtime semantics, these are not strenuous tests, certainly not as rigorous as a real-time system. Many real-time systems implemented in Ada are currently being developed and tested. These systems will provide the feed-back to Ada compiler vendors necessary for addressing real-time runtime problems.

There is not yet a single, widely accepted Ada runtime model that defines the Ada runtime environment interfaces and how the runtime environment should affect an Ada application or system. Some aspects of a runtime model are more amenable to standardization than others, but the convention of a single model would allow portability, reusability, distributability, fault-tolerance, and interoperability, all critical real-time issues. Runtime issues are by nature implementation-specific. For example, interfacing with UNIX is implementation specific, but UNIX is very widely used and UNIX standards are currently being developed. So any runtime issues which are difficult with respect to UNIX should be approached in a standard way. By addressing the issues *now* of defining a standard Ada runtime model and of targeting runtime issues which are difficult to implement, real-time system problems and portability and reusability problems can be avoided later.

## 2.2 Ada Runtime Systems

This section enumerates the runtime issues specified in the Ada Language Reference Manual[4] , describes potential runtime problem areas specific to Ada semantics, and explains the real-time/runtime requirements specific to the DAPSE project.

Although there is no standard Ada runtime model, it is generally agreed that a runtime system of an Ada compiler encompasses the following:

- the storage allocation scheme and garbage collection including the machine dependent data representation and data management in stacks and heaps during subroutines and tasks,

- the basic execution environment including tasking support, input/output, exception handling,
- the organization required to allow the Ada applications to interface with other languages, with specific hardware, and with the operating system.

When a compiler is intended to be part of the system, as in the case of an integrated programming environment, then the structure of the compiler, its interfaces, and its runtime requirements may be categorized runtime issues as well.

**Machine Dependencies** The Ada language definition allows for machine-dependencies in a controlled manner. Legal implementation-dependencies are outlined in Chapter 13 of the Ada Reference Manual [4], and must be specified in Appendix F of a given implementation. These include compiler implementation-dependent pragmas [2] and attributes, restrictions on representation clauses[3], interface to other languages, interrupts, the System package, the unchecked conversion and deallocation facilities, the calendar package, etc. An Ada compiler need not support these features in order to be validated, but systems-programs and real-time systems rely on these features for implementation.

### 2.2.1 Potential Problem Areas of Ada

Because Ada contains certain language features such as concurrency, exception handling, generics, and pragmas, all of which are not incorporated in any other established language, there is no established mechanism by compiler writers to support all of these features. Thus, one would expect that if there were problems with the runtime system that some combination of these features would be involved. Not only are there few runtime model conventions, but supporting distributed processing in Ada is more complicated that implementing a language with a simpler runtime model. So when Ada is operating in a UNIX environment, one would suspect that

[2] Pragmas provide an implementation with criteria for its selection of a mapping to the underlying machine.

[3] A representation clause is used to specify how the types of the language are to be mapped onto the underlying machine.

the Ada runtime, which supports Ada tasking and concurrency, could have schedule conflicts with the operating system kernel. These aspects of the runtime system of an Ada compiler are potential problem areas. Compiler-evaluation criteria will be directed toward these issues.

### 2.2.2 Runtime Requirements Specific to DAPSE

Timeliness, in particular a responsive user interface, is essential to a programming environment's ability to provide acceptable performance.DAPSE represents a class of real-time system which is targeted to the UNIX operating system and which has timing constraints measured in a human-oriented framework, on the order of tenth-of-seconds. This class is commonly referred to as "soft" real-time as opposed to "hard" real-time. A "hard" real-time system is typically embedded and has timing-constraints such that an interrupt must be handled on the order of milli-seconds. Both soft and hard real-time systems have timing constraints which must be met or the system is considered to fail.

DAPSE is systems-oriented, requiring access to system-dependent properties and services. It relies on the underlying operating system. For example, FENSTER (fenster means "window" in German), the DAPSE component which provides the graphical user interface, encapsulates device-specific graphics, as reported in [2]. The current implementation relies upon Sun work-station [4] graphics primitives. Hence, DAPSE requires the Ada runtime system to provide access to the UNIX provided SUN-Windows routines.

A FENSTER interface must be responsive not only to user input but also to output of the various components integrated in DAPSE, such as output from semantic analysis. FENSTER is best abstractly expressed in terms of Ada tasking, in which the display mechanism functions as a server. Runtime support for Ada tasking with synchronous I/O [5] is required by

[4]Sun Microsystems, Inc.

[5]Synchronous I/O refers to the communication between asynchronous processes which forces the processes to synchronize during the communication period, when one of the processes is accessing a limited resource such as an I/O device or a shared data structure. Synchronous I/O allows the processes to block; this type of system guarantees that data can be used immediately after an input operation and can be modified immediately after an output operation. Synchronous I/O is modeled in most high-level languages as it is easier to use. Asynchronous I/O refers to communication between asynchronous processes

FENSTER. For example, FENSTER must receive signals that the mouse has been clicked as well as signals that an error message should be displayed.

One of the components of DAPSE is ConCon (Configuration Control). ConCon as prototyped in DAPSE 2.0 is organized so that various users of DAPSE, each in a separate UNIX process typically on distinct machines, can automatically have their programming development efforts coordinated. Thus, DAPSE is a distributed program and therefore concurrency support is a DAPSE runtime issue. The underlying communications mechanism of the current implementation relies on the SUN RPC (Remote Procedure Call) layer which must be made accessible by the Ada runtime system.

DAPSE provides an Ada compiler for the user to compile the target program developed using DAPSE. The compiler is tightly integrated with the Ada structure editor as well as loosely integrated as a tool in the environment. Integration with another system is not normally considered a runtime issue unless the other system can contend for resources or operating system services. Thus, when integrating a compiler in a programming environment, its interfaces and resource constraints must be evaluated. For example, an Ada compiler is likely to have a memory management system which will support the runtime, including heap allocation and garbage collection in context of tasking and exception handling. The compiler memory management must be able to coexist with the DAPSE system.

As an integrated programming environment, DAPSE should allow the user (and programming tools) to have access to programming tools and other executables residing at the operating system shell with timeliness and reliability. For example, the compiler should be available from the editor and from the environment's "shell". DAPSE requires the runtime to provide an escape to the operating system shell in Ada semantics.

Finally, although the debugger is not considered part of the runtime environment (except perhaps in embedded systems where the debugger is installed on another machine), the debugger interacts with the runtime system. An argument can be made for including debugger evaluation in the compiler selection criteria; since the debugger can provide a good insight into the behavior of the runtime system, testing the debugger can be a good gauge for selecting a compiler based on the runtime system. In terms of

in which the processes can initiate an I/O operation and then continue execution. This type of system is useful in a concurrent system because computation and I/O can overlap.

the needs of DAPSE, the Ada debugger must be able to support testing the variety of features incorporated by DAPSE. In general, runtime problems in interfacing with the operating system are very difficult to diagnose, but, for example, the debugging of tasking should be supported.

This is not an exhaustive list of the Ada runtime requirements by DAPSE, but the more important runtime issues are included. Here is a synopsis of the runtime requirements:

- real-time performance ("soft");
- reliance on the underlying operating system (UNIX);
- Ada tasking, in particular, handling synchronous I/O;
- distributed processes in UNIX, with synchronous I/O;
- interfacing with the front end of the Ada compiler;
- escape to the operating system shell; and
- debugger supporting diagnosis of the above runtime requirements.

## 2.3 Our Experiences

This section describes selected runtime issues encountered in developing various components of DAPSE 2.0 using the Verdix compiler[6]. As a preface, the reasons why the Verdix compiler was selected for development are detailed.

**Why We Selected The Verdix Compiler** At the time (two years ago) the Verdix Ada compiler was selected for DAPSE 2.0 prototype development, there were three commercial Ada compilers targeted to SUN UNIX: the Telesoft, Alsys, and Verdix compilers. All three vendors supplied evaluation compilers. We based our evaluations of these on FENSTER and on the Ada structure editor. Not only was the Verdix compiler the most mature product at that time, featuring a symbolic debugger and the pragma

[6]Our experiences are based on the use of SUN-3 VADS (Verdix Ada Development System) 5.41 and 5.5.

interface to C, but it used DIANA (Descriptive Intermediate Attribute Notation for Ada) as the intermediate representation. This is convenient for integrating the compiler with the DAPSE prototype, one of the requirements of DAPSE 2.0. Moreover, DAPSE was already using the Verdix compiler based on an evaluation four years ago, and Verdix had demonstrated good communication and support.

**Ada Runtime Issues Encountered** Although our experiences are based on use of the Verdix compiler, general Ada runtime issues are discussed. These issues are illustrated by citing specific problems experienced using the Verdix compiler. The Ada runtime issues fall into the following categories:

- interfacing to UNIX;
- performance of the Ada scheduler;
- the pragma interface to C;
- interfacing to the front end of the Verdix compiler; and
- using the debugger.

The DAPSE problems mainly involved Ada and UNIX. There is no clear definition of UNIX in Ada semantics. General purpose UNIX targeted runtime systems are not finely tuned in the way that cross compilers for embedded real-time systems are. The types of problems reported by developers of embedded systems, as publicized by ARTEWG (Ada Run Time Environment Working Group), revolve around functional limitations of Ada semantics. Many real-time systems are hosted on UNIX, and therefore the interface to UNIX is a general Ada real-time/runtime problem.

### 2.3.1 Interface to UNIX

There is no standard of bindings from Ada to UNIX. Also there is no consistent mapping by the Verdix system onto the UNIX operating system calls. There is a subset of frequently used UNIX system calls in **standard** and in **verdixlib**. Since the pragma interface to C is supported and UNIX is written in C, there is an unofficial mapping to all UNIX calls, but

the consequences are unspecified[7]. For instance, shell variable values have proven to be inaccessible. The UNIX interface in the Verdix 5.5 release is an improvement to the 5.41 release.

### Tasking and the Ada Kernel

The most difficult problem to surmount is the relationship of the Ada tasking model to the UNIX processes. The Verdix runtime support for tasking [7, section 3.10] takes the only widely used approach taken for UNIX and Ada interaction, which is to implement the Ada kernel in one UNIX process. This has been provisionally adopted by POSIX Ada[8]. Thus, this is a general issue to be addressed, not a specific problem with the Verdix compiler.

The Verdix Ada compiler does not support synchronous I/O involving Ada tasks because the underlying mechanism for I/O relies on UNIX I/O facilities. The difficulty is due to the fact that if a UNIX process is waiting for I/O, the UNIX scheduler suspends the process. In the Ada/UNIX tasking model, the UNIX process containing the Ada task that is pending I/O also contains the Ada scheduler. This implies that the entire Ada application can become blocked. (See Figure 1.) Other proposals for mapping the Ada tasking model on UNIX processes are not feasible because of efficiency considerations.

Since the Verdix runtime does not support inter-tasking I/O, the application system must handle the problem. The most dangerous call, used frequently in the implementation of UNIX itself, is the UNIX *select* call. *Select* is used for I/O multiplexing, allowing blocking I/O. *Select* examines the I/O descriptors specified by the bit masks passed as parameters to see if they are ready for reading/writing, or have an exceptional condition pending. *Select* returns a mask of those descriptors which are ready. The last parameter is a pointer which specifies the maximum time-out to wait for the selection to complete. There are various errors which can result, such as EINTR if a signal is delivered before any of the selected events occur or

[7]The Verdix compiler is written in C. This possibly accounts for how few problems there are interfacing to UNIX.

[8]IEEE has formed a working group to develop a standard Portable Operating System Interface, based on UNIX, called POSIX. The 1003.5 working group, Ada interface to POSIX, has recently been formed to address Ada bindings.

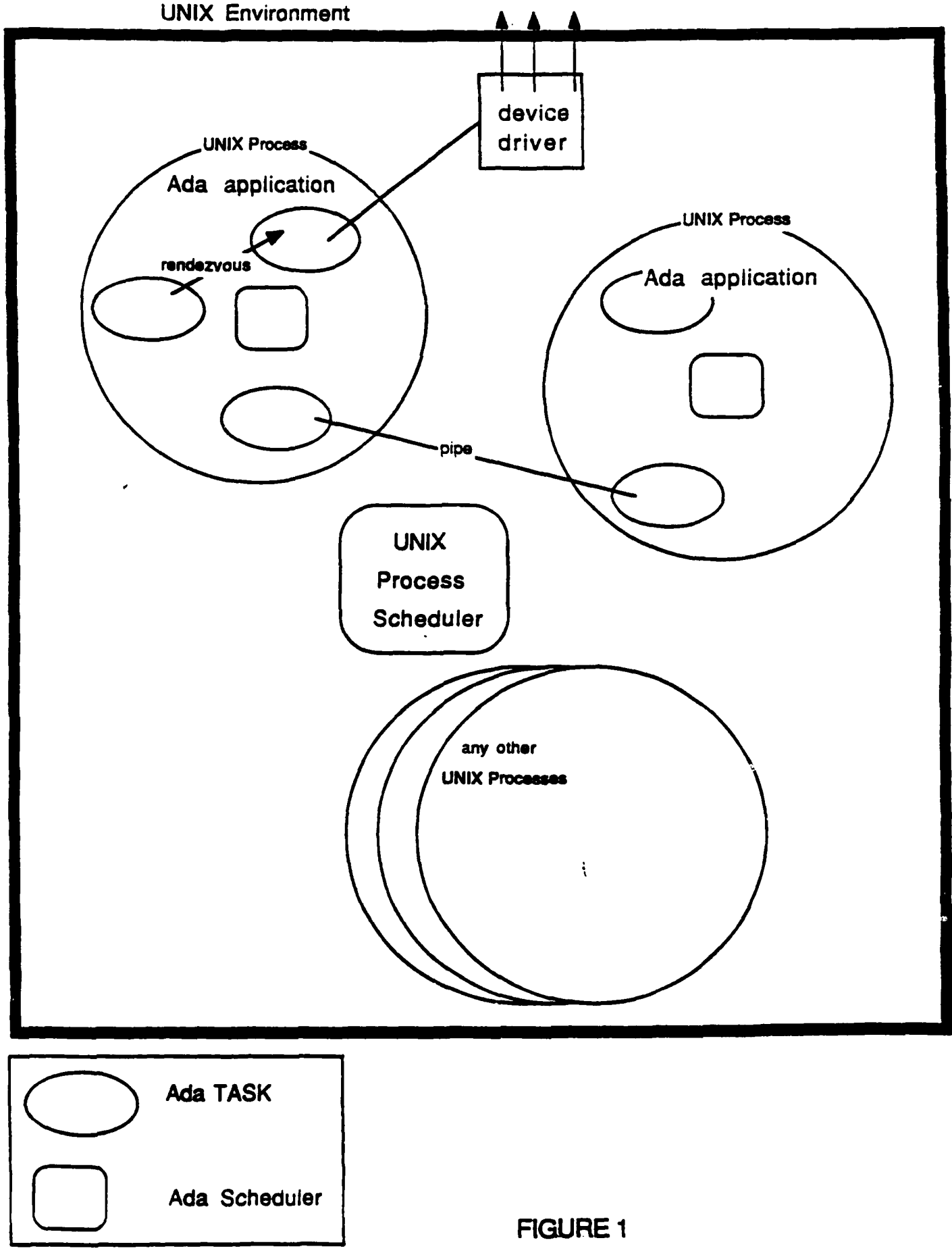

UNIX Environment
device driver
UNIX Process
Ada application
rendezvous
UNIX Process
Ada application
pipe
UNIX Process Scheduler
any other UNIX Processes
Ada TASK
Ada Scheduler

FIGURE 1

the time-out expires.

If the *select* call is made by one of the tasks in order to synchronize with another of the tasks operating from within the same UNIX task, the result is undefined. Sometimes deadlock ensues, although sometimes the scheduler functions properly. In fact the interface to the UNIX call *select* is unreliable. It is important to note that the underlying mechanism of the Verdix Ada scheduler relies on UNIX signals[9]. The Ada scheduler uses the SIGALRM signal to signal that an Ada task should wake up.

There are two alternative methods supporting synchronous I/O to investigate:

- With the Verdix compiler, it is possible to bind UNIX interrupts to a task entry using the implementation-dependent representation clause described in Chapter 13 of the Ada Reference Manual.

- Sockets can be used, with UNIX calls on the file-descriptor to control the pipe, and with an Ada **delay** to control the tasks.

When using a representation clause to map interrupts to UNIX signals, Verdix allows interrupts to be set up like a task entry rendezvoused with an accept. This alleviates the problem of the deadlocking *select* statement but is limited by the number of signals provided by UNIX. The Ada tasking is thus governed by the limitation of UNIX signals : There are 32 signals, only 2 of which are user definable. Because signals do not take arguments. a signal catcher is matched to an argumentless task entry. However, one would like to be able to discriminate between different instances of causing the same signal. The semantics do not specify which of the blocked tasks has caused the signal and should now be unblocked.

FENSTER, the user interface, uses both of the above alternatives to communicate with the UNIX environment, the "outside world". The UNIX socket method is used in the DAPSE 2.0 Distribution Toolkit prototype involving communication among distributed UNIX processes. This has generally been successful. However, this, like any method of communication, is very difficult to diagnose and monitor. Furthermore, there is semantic ambiguity about the interaction of the UNIX process scheduler(s) and the Ada kernel scheduler. The RPC mechanism used in the Distribution

[9]POSIX advises that an Ada program which uses signals needed by the runtime for internal scheduling should be considered "unsafe".

Toolkit makes assumptions about the UNIX runtime environment, and the *delay* statement makes assumptions about the Ada runtime environment.

How should Ada tasking with synchronous I/O be supported in the UNIX environment? The desirable solution would be for the runtime system to provide reliable I/O involving tasks so that each application system implementor does not have to determine the conflicts of that runtime system with UNIX.

### 2.3.2 Interfacing to C

Because the UNIX operating system is written in C, the pragma interface to C can be used to access UNIX calls or UNIX library routines that are not provided by the Ada runtime system. There are some problems in the interface to C provided by the Verdix compiler.

For example, an interface to the UNIX *system* call is not provided by Verdix and so must be accessed by the interface to C. The function *system* takes a string argument and returns an error status code of 0, 1, or 2 reflecting whether there is no error, an anomalous error, or output directed to standard error. The string argument contains the name of an executable available from the Bourne shell. When executed from a program a separate UNIX shell process is invoked and runs the executable specified by the string argument. This mechanism does not work properly from Ada. The status returned is always 0 even if an error condition has been raised. But the status is necessary to determine whether the current value for the error condition is valid. Since the status must be relied upon to indicate the success of the *system* call and the interface to C cannot provide access to the status shell variable, the interface to C is insufficient for access to the operating system.

The implications of the weakness of the *system* call are disturbing because it demonstrates the weakness of the general interface to the operating system. An Ada runtime system cannot possibly provide all UNIX routines. Therefore a reliable general purpose mechanism such as the pragma interface to C must be provided by the runtime system. Moreover, a mechanism such as *system* which makes executables residing at the shell available, should be provided by the runtime system in Ada semantics.

### 2.3.3 Issues of the Ada Scheduler

Besides the problems encountered in coordinating the Ada scheduler and the UNIX scheduler, there are other issues which have arisen involving the Ada scheduler.

#### Critical Regions to Support Multi-Language Interface

The Verdix runtime system does not provide a mechanism to protect a critical region when interfacing with a language other than Ada. This implies that memory allocation performed during an interface to another language is unsafe because there is no method of preventing an interrupt by the Ada task scheduler during which another memory allocation by another task could take place[10]. This is an important issue because memory allocation has far-reaching consequences in terms of reliability.

This is a weakness in the Ada/UNIX interface definition, rather than specifically a problem case of the Verdix compiler. In the interests of reusability of existing real-time systems, publicly available UNIX functions, data-base managers, generator-tools, etc., it is necessary to provide a general mechanism written in Ada semantics for protecting critical regions. Without such a mechanism, existing code written in other languages cannot be used with reliability. This is an issue of practicality. Keep in mind that DAPSE, like some other real-time systems, is hosted on UNIX and that there is much existing code of/on UNIX written in C. This code must be accessible from the Ada applications without compromising the consistency and reliability of that code.

**Proposed Solutions:** One viable solution for Verdix to protect memory allocation would be to provide Ada units which interface to modified UNIX functions, in which the interrupts are disabled around the memory allocations. Of course, the runtime system would also need to use these memory allocation routines. This would not be a general purpose solution, however. A better solution would be for the runtime system to provide two general purpose routines: *protect* and *de-protect* which could be called around any foreign-language routines deemed critical or sensitive. These

[10]Memory allocation in Ada is an atomic operation.

routines would effectively disable/enable the scheduler. Their implementation (bodies) could be changed in future releases of the compiler without affecting their dependents. The provision of a mechanism such as this would increase the reliability, generality, extensibility, and robustness of any Ada runtime system.

This solution is controversial because it can be argued that it could allow dangerous situations. It is a good approach for a cooperative system, but not for a contentious system. For instance, if interrupts can be disabled in this way by a task, then that task can preempt others. This is a good general purpose approach for supporting critical regions provided that the mechanism is used carefully and that imported compilation units with tasks are re-used carefully.

### Pragma Task Entry Call

The Distribution Toolkit relies on the SUN RPC layer to provide a communications toolbox to support distribution, accessible by the pragma interface to C. One of the issues encountered in DAPSE when interfacing Ada with C is that the complete semantics of C could not be utilized. A pointer to a function can be passed as a parameter in C. When using a SUN RPC function (written in C), a distributed application would like to pass an entry as a parameter. However, the Verdix Ada scheduler does not provide handles to the Ada tasks, so an Ada entry cannot be accessed from a C function. In order to truly interface C and Ada, the scheduler should provide a signature to an entry which would enable the provision of a *pragma Task Entry Call* .

### Combinations of Ada Features

There was an instance during DAPSE 2.0 development (a prototype of the Distribution Toolkit, an application using ConCon) when deadlock occurred. It was not obvious what happened but the Sun RPC library interacted poorly with the Ada (Verdix release 5.41 ) runtime tasking model. It involved deeply nested generic packages with task types; in an instance of the outermost generic, one task called a SUN RPC function. When the packages where made non-generic, there was only an occasional block. When other packages were imported, the blocking recurred. The exact nature of the problem was not analyzed since the thrust of our investigation

is into a prototype rather than analysis of runtime anomalies. As described above in the section on Tasking and the Ada kernel, we currently use the UNIX sockets and Ada *delay* approach, so the problems of this particular combination of Ada features is no longer an issue.

There were other instances when the combination of various components, each involving Ada tasking or independent memory management schemes, and each functioning properly independently, blocked indefinitely. It is clear that these examples are legal Ada, but are unlikely to have analogues in the ACVC. This type of artifact should be tested when evaluating the runtime system of an Ada compiler.

### Limitations of Ada Semantics

During implementation of DAPSE 2.0, there were some complaints that the setting of priorities is too static and that the Ada scheduler behaves in an undemocratic fashion. It is not possible to dynamically change the priority of a task. The assignment of different priorities to modify the behavior of the scheduler is erroneous Ada because not all Ada compilers support multiple priorities. The ARTEWG is addressing both the limitations of Ada semantics with respect to real-time systems and the feasibility of a portable runtime. A portable runtime would enable an application to supply its own scheduler.

## 2.3.4 Interface to the Verdix Compiler

Integration of the DAPSE Ada structure editor with the Verdix compiler is effected by transforming the representation of DIANA, the intermediate form of Ada employed by the DAPSE Ada editor, to the DIANA representation utilized by the Verdix compiler. The fact that Verdix employs DIANA at all is incidental and fortunate to the mechanism of integrating the Verdix compiler with DAPSE. Although Verdix employs DIANA, there were unforeseen problems with the integration related to the Verdix memory management scheme.

Both the Verdix Compiler and the DAPSE persistent object base manager, generated by DTB (DaTa Base), are written in C. DTB has its own memory management scheme that uses standard C/UNIX memory allocation routines. The Verdix memory management scheme is used by the

runtime environment and thus supports allocation/deallocation in stacks and heaps; it also enables an extremely efficient reading/writing of the parse tree (the Verdix DIANA tree) for supporting the separate compilation feature of Ada. The Verdix memory management scheme is implemented by renaming the standard UNIX allocation routines thus ensuring that all memory allocation is managed by the Verdix memory manager. However, the archives provided by the Verdix library are incomplete and in some cases organized differently than the standard UNIX library. For example, *realloc* is defined to be an abort, terminating execution, and *calloc* is defined in malloc.o rather than in calloc.o. And *cfree* is not defined at all. Another inconsistency is that *system* is defined to be a global variable which interferes with the C function *system* called by DAPSE. There are other naming collisions because many Verdix C functions which should have been declared static (local) are not. The memory-management does not support standard UNIX libraries of functions at runtime. Without revision the implementation of the Verdix front end would cause linkage failures and runtime problems in the context of other standard C/UNIX binaries.

#### Analyzing the Implementation of an Ada Compiler in Context of an Integrated Programming Environment

Analyzing the quality of the source code which implements an Ada compiler is hardly a conventional approach to evaluating the runtime characteristics of a compiler. Besides, there are no standard measures of code quality. But assessing source code quality subjectively is encouraged when incorporating any piece of code in a system. And therefore it is necessary to examine the source of a compiler when integrating it with a programming environment. We fixed the linkage failures by modifying the Verdix interface source code. However, the runtime problems imposed by the memory-management scheme of the Ada compiler on the entire DAPSE system were not easily modifiable by DAPSE.

### 2.3.5 Using the Debugger

Since the debugger interacts with the runtime system, it is possible to evaluate a compiler's runtime system by testing some features of the runtime

system with the debugger. I will give an example of a problem with the Verdix 5.41 debugger which is fixed in the Verdix 5.5 debugger to illustrate the importance of the debugger with respect to the runtime system. Interrupt driven code is impossible to debug with versions of the Verdix debugger prior to Verdix 5.5. When using the Verdix 5.41 debugger, if a UNIX signal occurs *gx* can be typed by the user to pass the signal to the program and continue execution. But for example, the signal SIGIO is flagged every 1/15 of a second by SunView to indicate that the mouse has not moved. Since it is impossible for a person to type *gx* before the next SIGIO arrives, the debugging session cannot proceed. With the Verdix 5.5 debugger, it is possible to specify that a particular UNIX signal should automatically be passed on to the program, which allows the debugging session to continue.

This example demonstrates that the runtime system features should be supported in the debugger. In general it is very difficult to diagnose runtime problems. The debugger could support the operating system interface and certain Ada features such as tasking and generics. The placement of breakpoints in separate generic instantiations is an issue. The user should be able to specify whether a breakpoint should be put in one or all instantiations.

## 2.4 Compiler Evaluation

This section proposes an approach for evaluating the runtime performance of an Ada compiler and for formulating questions about the features of an Ada runtime system including the Ada/UNIX interface problem.

### 2.4.1 Domain Specific Sample

The runtime requirements of the applications system should be summarized. The particular systems services, such as I/O, network services, relational data-base services, etc., that will be required should be tested for reliability. A sample program should be devised which includes these features in order to test and compare the various Ada compilers under consideration. In our case, we would test a compiler runtime with FENSTER, because it is a multi-tasking system which interacts with the UNIX operating system services, performs I/O, has generics, has moderate scheduling

demands, and must respond in ("soft") real-time. The domain test should not be just a subjective test with shades.

Here are two examples of how the Ada runtime did not entirely support FENSTER. FENSTER caused one pre-release of the 5.5 Verdix compiler to abort at runtime. This was due to the fact that the compiler expected arguments passed to task entries to be 4-byte quantities whereas the 5.41 release did not make that assumption. Once Verdix fixed this particular bug, we were very pleased with the improved runtime performance attributable to the new runtime support for tasking. This bug illustrates the nature of runtime semantics which are not exercised by the ACVC test suites. This story illustrates the fact that feed-back from real-time systems currently under development is essential for maturing the Ada runtime technology. Furthermore, communication and a good working relationship with a compiler vendor, such as the DAPSE project with Verdix, should also be considered in the selection of a compiler for a large real-time project development.

The Verdix Ada runtime also does not support FENSTER when integrated with a library unit which links the Verdix compiler front-end[11]. The point is that FENSTER is a good domain test for our application. Producing a domain sample test is appropriate when selecting a compiler from among available compiler choices.

### 2.4.2 Combination of Ada Features

Besides domain specific features, there are also general runtime features which should be tested with sample tests which are more strenuous than the ACVC suite. A good test is a combination of the following features:

- concurrency;
- exceptions;
- generic packages; and
- interface to the operating system

[11]Since much of the archive which supports the front-end also supports other phases of the compiler, it would be time-consuming to determine the cause of the conflict. Because the compiler integration experiment targets other issues, we did not fully investigate what causes the blocking. Instead we placed the library unit which links the Verdix front-end into a separate process and proceeded with the direct investigations of the experiment.

e.g. a task with nested generics and which makes a system call.

### 2.4.3 Formulate Questions

When assessing the runtime system of an arbitrary Ada compiler, a specific series of questions should be asked. There are general questions that can be asked about the runtime system, but it must be emphasized that the particular features required of the runtime system depend on the application. Based on our experience, questions should be asked about the compiler which enable one to assess the tasking runtime model, the interface with UNIX (or the operating system), and the pragma interface to C if the target is UNIX.

Consider these examples :

- Does the compiler have a reliable Ada interface to UNIX system calls, not just pragma interface to C, but tested routines supporting good Ada style? Will the compiler vendor maintain these Ada routines? Do these Ada routines abide by the POSIX standards? In particular, does the UNIX *select* call function without disabling the Ada scheduler?

- Does the compiler provide synchronous I/O among Ada tasks? If so, how reliable is the mechanism, and how is this implemented by the UNIX operating system?

- Which implementation dependent features defined in Chapter 13 of the Ada Reference Manual are included? Every real-time system is by nature target-implementation dependent; how will the compiler runtime support this dependency? We could not have managed without pragma interface to C and representation clauses. The interface to C is a critical issue when evaluating an Ada compiler's runtime system for UNIX-hosted real-time applications, because operating system code is written in C as well as much existing code with which the application will need to interface.

- How are these implementation dependent features supported? For example, is there a mechanism to protect critical regions when interfacing with foreign languages? Does the scheduler provide handles, such as implementation defined attributes, to the tasks and task-entries, so that they can be accessed by C/UNIX system routines?

- Does the compiler provide a general mechanism in Ada semantics to escape to the operating system shell and run executables?

- How does the Ada scheduler behave? How modifiable is the scheduler? Can priorities be changed dynamically? If so, systems which take advantage of extra features supporting tasking cannot be classified as legal Ada programs[12].

- How does the debugger support diagnosis of runtime issues such as exception handling and I/O? Can specified signals be trapped or ignored? Can I/O which is normally performed during program execution be redirected? Can output not normally performed during program execution be properly printed? The debugger interacts with the runtime system, and as such a compiler may be evaluated in terms of the performance of its debugger. The user-interface of the debugger has the most impact on its usefulness, so the user-interface should figure prominently in this evaluation.

- If the compiler is to be integrated with the system, then what are the runtime requirements of the Ada compiler? How is memory management performed? How modifiable is the source code of the Ada compiler?

- Does the runtime system support performance analysis in any way? Is there a compile-time option to insert performance measuring code?

- What methods of storage allocation for access variables is used? Does unchecked-deallocation free heap storage? Is there garbage collection[13]?

There are many other guidelines which we have not considered, such as whether the runtime overhead is incurred in those objects which do not take advantage of execution features such as exception handling and tasking. This affects the size of the executable, important for real-time systems.

[12] ARTEWG recommends a variety of scheduling options and user-defined interfaces to their implementations.

[13] As yet, no Ada compilers support garbage collection.

## 2.5 Conclusions and Recommendations

In summary, it is important to keep in mind that the Ada compiler runtime technology is immature and that now is the time to address those runtime issues supporting systems-oriented programs and real-time systems. Very little information about Ada real-time runtime experiences has been disseminated. DAPSE represents a class of real-time systems hosted on UNIX. Therefore observations about the DAPSE Ada runtime experiences reveal Ada runtime issues which are difficult to implement and universal. These runtime issues should be considered when selecting an Ada compiler runtime environment.

In the absence of well-defined runtime system criteria, it is imperative to define the application system requirements on the runtime system, and to address these requirements when formulating questions about a compiler's suitability. Based on the DAPSE experience with Ada distributed "soft" real-time runtime issues, we conclude that the majority of issues are application dependent and operating system (UNIX) specific.

We emphasize addressing the following Ada features in a UNIX targeted application:

- the ability to reliably integrate Ada tasking with UNIX process control, including the provision of synchronous I/O in the Ada tasking model;
- the provision of a full and reliable interface with the UNIX operating system, and the adoption of the POSIX standards;
- a full and reliable interface to other languages, particularly with C;
- the provision of reliable tasking control including the ability to specify critical regions when interfacing with other languages and the ability of the Ada scheduler to provide handles to Ada entries.

Issues which are general Ada and real-time problems but not specifically UNIX problems include:

- the ability to control the behaviour of the Ada scheduler with optional and dynamic priorities and attributes;
- the ability to diagnose runtime behaviour with the debugger.

Addressing the definition of a standard Ada runtime model and targeting runtime issues which are difficult to implement will avoid real-time system problems and portability and reusability problems later.

# Bibliography


[1] M. Marcus, S.C. Schaffner, K. Sattley, G. Albert,
DAPSE: A Distributed Ada Programming Support Environment,
In Proceedings of the IEEE 2nd International Conference of Ada Applications and Environments, April 1986.

[2] Stuart C. Schaffner, Martha Borkan,
SEGUE: Support for Distributed Graphical Interfaces,
In Proceedings of the 21st Hawaii International Conference on System Science, January, 1988.

[3] C. Mugur Stefanescu, H. Dieter Rombach,
A Development Methodology for Distributed Ada Applications.
In Proceedings of the Fifth Washington Ada Symposium, Ada: The State of the Technology, June 1988.

[4] Reference Manual for the Ada Programming Language,
ANSI/MIL/STANDARD, U.S. Department of Defense, February 17, 1983.

[5] Ada: Language, Compilers and Bibliography, edited by M.S. Rogers, Cambridge University Press, Cambridge, 1984.

[6] Draft 12, Portable Operating System Interface for Computer Environments, IEEE Technical Committee on Operating Systems, October 12, 1987.

[7] Verdix Ada Development System, VADS UNIX Implementation Reference, (Including Ada RM appendix F), VADS version 5.41.